

# SLICING CONCURRENT LOGIC PROGRAMS

JIANJUN ZHAO and JINGDE CHENG and KAZUO USHIJIMA

*Department of Computer Science and Communication Engineering*

*Kyushu University*

*Hakozaki 6-10-1, Fukuoka 812-81, Japan*

E-mail: {zhao,cheng,ushijima}@csce.kyushu-u.ac.jp

## ABSTRACT

Program slicing has been widely studied for imperative programs, but for logic programs it is just starting. In this paper we address the problem of slicing concurrent logic programs. To solve this problem, we propose three types of primary program dependences between arguments in concurrent logic programs, named the *sharing dependence*, *communication dependence* and *unification dependence*. We also present a new program representation named the *argument dependence net* (ADN) for concurrent logic programs to explicitly represent three types of primary program dependences in the programs. Based on the ADN, we formally define various notions about slicing concurrent logic programs and show that our slicing algorithm can produce static slices for concurrent logic programs at argument level.

## 1. Introduction

Program slicing, originally introduced by Weiser [21], is a decomposition technique which extracts from program elements related to a particular computation. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. Slices can be classified in two categories: a *backward slice* with respect to a set of program elements  $E$  consists of all program elements that might affect the values of the variables at members of  $E$ , and a *forward slice* with respect to  $E$  consists of all program elements that might be affected by the computations performed at members of  $E$ . Also, slices can be divided into: *static slices* that are computed without making assumptions regarding a program's input, and *dynamic slices* whose computations rely on a specific test case. Moreover, it is necessary to understand two different but related slicing problems: a *slice is executable* in the sense that the slice of a program with respect to a program point  $p$  and variable  $v$  consist of a reduced program that computes the same sequence of values for  $v$  at  $p$ , and a *slice is not necessarily executable* in the sense that the slice of a program with respect to program point  $p$  and variable  $v$  consists of all elements of the program that might affect the value of  $v$  at point  $p$ . Program slicing has been widely studied in the context of imperative programming languages. In such languages, slicing is typically performed by using *program dependence graphs* [4, 6, 9, 14]. Program slicing has showed great usefulness for many software engineering activities such as program understanding, debugging [10], testing [1], maintenance [7] and complexity measurement [2, 14]. For

more information, see Tip’s nice survey on program slicing of imperative programs [18].

However, although program slicing has been widely studied for imperative programs, research on slicing logic programs is just starting. To our knowledge, only three articles deal with the problem of slicing logic programs, and no slicing algorithm for concurrent logic programs exists until now, which can produce slices at argument level <sup>a</sup>.

Gyimóthy and Paakki [8] proposed a specific slicing algorithm for sequential logic programs in order to reduce the number of queries of an algorithmic debugger. They defined a slice of a logic program as a subtree of the proof tree of the program, and used the program dependence graph to compute slices. Their definition of a slice is too specific to give a general view of the slice, and only used oriented data dependence information to compute the slice. Schoenig and Ducassé [15] proposed a backward slicing algorithm for Prolog which produces executable slices. For a target Prolog program they defined a slicing criterion which consists of a goal of  $P$  and a set of argument positions  $P_a$ , and a slice  $S$  as a reduced and executable program derived from  $P$ . The algorithm can only be applicable to a limited subset of Prolog programs. Unfortunately, it is not straightforward to use the existing slicing algorithms for sequential logic programs to compute slices for concurrent logic programs. First, these algorithms did not consider the problem of how to handle the guard parts of clauses in concurrent logic programs. Second, the static analysis techniques they adopted for dependence analysis of sequential logic programs [3, 5] cannot be applied to concurrent logic programs, where no assumptions can be made about the order of execution of the goals or about the interleaving of their reduction.

In [22, 23] Zhao *et al.* defined some static and dynamic slices of concurrent logic programs and computed them based on a dependence-based representation called the *literal dependence net* (LDN). Since the literal dependence net can only be used to represent program dependences between literals and not between arguments in the programs, the slicing algorithms based on it can not produce sufficiently precise slices with abstracted arguments.

In this paper we address the problem of slicing concurrent logic programs. To solve this problem, we propose three types of primary program dependences between arguments in concurrent logic programs, named the *sharing dependence*, *communication dependence*, and *unification dependence*. We also present a new program representation named the *argument dependence net* (ADN) for concurrent logic programs to explicitly represent the three types of primary program dependences in the programs. Based on the ADN, we formally define various notions about slicing concurrent logic programs and show that our slicing algorithm can produce static slices for concurrent logic programs at argument level.

---

<sup>a</sup>By “argument level”, we mean that can abstract away arguments inside literals in concurrent logic programs.

The rest of the paper is organized as follows. Section 2 defines some notions about slicing concurrent logic programs. Section 3 briefly introduces the concurrent logic programming language KL1 and defines some basic notations and the parallel control-flow net for concurrent logic programs. Section 4 defines three types of primary program dependences between arguments in concurrent logic programs and presents the argument dependence net for concurrent logic programs. Section 5 shows how to compute slices of concurrent logic programs based on the argument dependence net. Concluding remarks are given in Section 6.

## 2. Issues on Slicing Concurrent Logic Programs

Program slicing has been widely studied in the context of imperative programs. A major difference between logic programs and imperative programs in this regard is that logic programs are nondeterministic in general and lack of explicitly control flow and data flow of the programs. Because of this behavior, to capture control flow and data flow in logic programs is significantly complex. As a result, slicing for logic programs is considerably more difficult than for imperative programs, and it cannot use slicing algorithms for imperative programs to compute slices of logic programs.

To solve the problem of program slicing, two key issues must be considered, that is: (1) how to define a slice of a program and (2) how to compute the slice correctly for the program in term of the definition. Although different motivations of slicing could lead to different views to define a slice, to compute a slice of a program correctly, it is necessary to make both control flow and data flow explicit in the program.

The problem of slicing concurrent logic programs in this paper is originally motivated by the development of a support environment for program analysis of concurrent logic programs [22, 23]. The system consists of a group of software engineering tools to support program understanding, debugging, testing, maintenance and complexity measurement for concurrent logic programs. In program understanding of concurrent logic programs, we usually want to know which literal might affect a literal of interest, and which literal might be affected by the computation of a literal of interest in the program. There are similar requirements during the maintenance of concurrent logic programs when a literal of the programs has been modified. In debugging we are interested in finding the literals of the program that are responsible for an incorrect result of the program's execution. As a result, two kinds of information is basically required in such a system:

1. *Which literal in a clause of a concurrent logic program might be affected by a literal in another clause of the program ? and*
2. *Which literal in a clause of a concurrent logic program might affect a literal in another clause of the program ?*

Since program slicing is a useful technique which can produce such information,

we would like to develop a slicer for concurrent logic programs and embed it into the system to provide sufficient information for the requirements.

Based on above considerations, in this paper, we focus our attention on whether or not a slice is adequate to provide sufficient information for various requirements rather than it is executable.

In the following we define some notions of slices for concurrent logic programs. To abstract away arguments inside literals, we first define the notion of a reduced literal of a concurrent logic program.

**Definition 2.1** *A reduced literal of a literal  $l$  in a concurrent logic program is a literal  $l'$  that is derived from  $l$  by replacing zero, or more arguments of  $l$  with anonymous variables.*

According to the above definition, a literal has at least one reduced literal, the literal itself. Having the definition of a reduced literal, some notions of a static backward and forward slice of a concurrent logic program can be described as follows:

- A *static slicing criterion* for a concurrent logic program  $P$  is a 2-tuple  $(l, a)$ , where  $l$  is a literal in  $P$  and  $a$  is an argument of  $l$ .
- A *static backward slice*  $SBS_P(l, a)$  of a concurrent logic program  $P$  on a given static slicing criterion  $(l, a)$  consists of all reduced literals in  $P$  that might affect the value of variables in argument  $a$  at  $l$ .
- A *static forward slice*  $SFS_P(l, a)$  of a concurrent logic program  $P$  on a given static slicing criterion  $(l, a)$  consists of all reduced literals in  $P$  that might be affected by the value of variables in argument  $a$  at  $l$ .

Notice that unlike Gyimóthy and Paakki [8] who defined a slice of a sequential logic program as a piece of proof tree of the program, and Schoenig and Ducassé [15] who defined a slice of a Prolog program as an executable program, we defined a slice of a concurrent logic program as a set of literals of the program that is not necessary executable. This is mainly because that their purpose to slicing sequential logic programs is to aid bug location during debugging, which is only a part of our purpose. A slice that is defined as a set of literals of a concurrent logic program could provide us sufficient information to the problems of understanding, maintenance, complexity measurement and even debugging of concurrent logic programs.

### 3. Preliminaries

#### 3.1. KL1

We assume that readers are familiar with the basic concepts of logic programs, and in this paper, we will use KL1 [19], a concurrent logic programming language based

on Guarded Horn Clauses (GHC), as our target language. This language illustrates the basic mechanisms of concurrent logic programming.

A *term* is a variable, a constant, or a compound term  $f(t_1, \dots, t_n)$  where  $f$  is an  $n$ -ary function symbol and the  $t_i$  are terms,  $1 \leq i \leq n$ . An *atom* is of the form  $p(t_1, \dots, t_n)$ , where  $p$  is an  $n$ -ary *predicate* symbol and the  $t_i$  are terms called *arguments*,  $1 \leq i \leq n$ . A *literal* is either an *atom* or the negation of an atom. The number of arguments of a literal is called its *arity*.

A *guarded clause* is a formula of the form:  $H :- G_1, G_2, \dots, G_n | B_1, B_2, \dots, B_m$ . ( $m, n \geq 0$ ), where  $H, B_1, B_2, \dots, B_m$  are literals, and  $G_1, G_2, \dots, G_n$  are guard test predicates.  $H$  is called the head of the clause,  $G_1, G_2, \dots, G_n$  are called the *guard* of the clause, and  $B_1, B_2, \dots, B_m$  are called the *body* of the clause, respectively. “:-”, read *if*, denotes implication, and “|” is called the *commit operator*. If the guard is empty the clause is written as:  $H :- B_1, B_2, \dots, B_m$ . and the commit operator is omitted. If the body is empty and the guard is not empty, the clause is written as:  $H :- G_1, G_2, \dots, G_n | true$ . And if both the guard and the body are empty the clause is called a *unit clause*, and is written simply as:  $H$ . A clause whose body includes exactly one goal is called an *iterative clause*. The clause with only negative literals is referred to as a *goal*. A *procedure* is a set of clauses each of which has the same predicate name and arity. A *KL1 program* is a finite set of guarded clauses.

Figure 1 shows a sample KL1 program called **STACK** which implements a stack function. The stream of the first argument of procedure **stack** receive the list of the stack commands, and the stream of the second argument output the results. The stack is maintained in the stream of the third argument. The possible input commands are: **push(D)** pushes the value of D into the top of the stack; **pop** gets the value from the top of the stack and outputs it; **pop(N)** gets N values from the top of the stack and outputs them as a list with the length N; **reverse(N)** gets N values from the top of the stack and outputs them as a reverse list with the length N.

### 3.2. Operational Semantics

Concurrent logic programming languages apply a new reading of logic programs, i.e., the *process reading* [16]. According to this reading, each goal atom is viewed as a process, the goal as a whole is viewed as a network of concurrent processes, whose process interconnection pattern is specified by the logical variables shared between goal atoms. Processes communicate by instantiating shared variables and synchronize by waiting for some logical variable to be instantiated.

We introduce an operational semantics of a concurrent logic program informally according to the reference [17]. The sketch of operational semantics uses two types of processes, an *And-process* and an *Or-process*.

The goal statement is fed to a root-process, a special case of an Or-process. Given a conjunction of goals, a root-process creates one And-process for each goal. When

```

C1: stack([push(D)|I],O,S):-
    stack(I,O,[D|S]).
C2: stack([pop|I],O,S):-
    O=[A|NO], pop(A,S,NS), stack(I,NO,NS).
C3: stack([pop(N)|I],O,S):-
    O=[L|NO], pop(N,L,S,NS), stack(I,NO,NS).
C4: stack([reverse(N)|I],O,S):-
    O=[L|NO], rev:rev(R,L), pop(N,R,S,NS), stack(I,NO,NS).
C5: stack([],O,_):-
    O=[].
C6: pop(A,[X|S],NS) :-
    A=answer(X),NS=S.
C7: pop(A,[],NS):-
    A=empty, NS=[].
C8: pop(N,L,S,OS) :-
    N>0 | L=[X|NL], pop(X,S,NS), NN=N-1, pop(NN,NL,NS,OS)
C9: pop(0,L,S,OS) :-
    L=[], OS=S.

```

Figure 1: A sample KL1 program.

all these And-processes succeed, the root-process succeeds. When one of these fails, it fails.

Given a goal  $g$  with the predicate name  $p$ , an And-process creates one Or-process for each clause defining the predicate  $p$  and passes the goal to each process. When at least one of these Or-processes succeeds, the And-process commits itself to the clause sent to that Or-process, and aborts all the other Or-processes. Then it creates an And-process for each goal in the body part of the clause and replaces itself by these And-processes. It fails when all these Or-processes fail.

Given a goal and a clause, an Or-process unifies the goal with the head of the clause and solves the guard part of the clause by creating an And-process for each goal in the guard. When all these And-processes succeed, it succeeds. If one of these fails, it fails.

### 3.3. Parallel Control-Flow Nets

In order to make both control flow and data flow explicit in concurrent logic programs, we present two graph-theoretical representations named the *parallel control-flow net* and *augmented parallel control-flow net* to explicitly represent control flows and argument information in concurrent logic programs. The definition of the parallel control-flow nets follows the operational semantics of a concurrent logic program introduced above.

To represent a concurrent logic program using an arc-classified digraph, we distinguish each argument, literal, and clause of the program. We assume that the clauses of a concurrent logic program are denoted by  $C_1, C_2, \dots, C_n$ . The literals (including the guard test predicate) of a clause are numbered from left to right, such that the head is numbered by 0, the guard test predicate or the first body literal (if the guard is empty) is numbered by 1, and so on. The arguments of a literal are numbered from left to right by 1, 2, .... If  $C_i$  is a clause of a concurrent logic program  $P$ , the position of the  $k^{th}$  argument of the  $j^{th}$  literal is uniquely defined in  $P$  by the tuple  $(C_i, j, p, k)$  such that  $p$  is the predicate name of the  $j^{th}$  literal of  $C_i$ . Such a tuple is called an *argument position*. The definition of argument positions is taken from [3].

**Definition 3.1** A digraph is an ordered pair  $(V, A)$ , where  $V$  is a finite set of elements called vertices, and  $A$  is a finite set of elements of the Cartesian product  $V \times V$ , called arcs, i.e.,  $A \subseteq V \times V$  is a binary relation on  $V$ . For any arc  $(v_1, v_2) \in A$ ,  $v_1$  is called the initial vertex of the arc and said to be adjacent to  $v_2$ , and  $v_2$  is called terminal vertex of the arc and said to be adjacent from  $v_1$ . A predecessor of a vertex  $v$  is a vertex adjacent to  $v$ , and a successor of  $v$  is a vertex adjacent from  $v$ . The in-degree of vertex  $v$ , denoted by  $in-degree(v)$ , is the number of predecessors of  $v$ , and the out-degree of a vertex  $v$ , denoted by  $out-degree(v)$ , is the number of successors of  $v$ . A simple digraph is a digraph  $(V, A)$  such that no  $(v, v) \in A$  for any  $v \in V$ .

**Definition 3.2** An arc-classified digraph is an  $n$ -tuple  $(V, A_1, A_2, \dots, A_{n-1})$  such that every  $(V, A_i) (i = 1, \dots, n-1)$  is a digraph and  $A_i \cap A_j = \emptyset$  for  $i = 1, 2, \dots, n-1$  and  $j = 1, 2, \dots, n-1$ , and  $i \neq j$ . A simple arc-classified digraph is an arc-classified digraph  $(V, A_1, A_2, \dots, A_{n-1})$  such that no  $(v, v) \in A_i (i = 1, \dots, n-1)$  for any  $v \in V$ .

**Definition 3.3** A path in a digraph  $(V, A)$  or an arc-classified digraph  $(V, A_1, A_2, \dots, A_{n-1})$  is a sequence of arcs  $(a_1, a_2, \dots, a_l)$  such that the terminal vertex of  $a_i$  is the initial vertex of  $a_{i+1}$  for  $1 \leq i \leq l-1$ , where  $a_i \in A (1 \leq i \leq l)$  or  $a_i \in A_1 \cup A_2 \cup \dots \cup A_{n-1} (1 \leq i \leq l)$ , and  $l (l \geq 1)$  is called the length of the path. If the initial vertex of  $a_1$  is  $v_I$  and the terminal vertex of  $a_l$  is  $v_T$ , then the path is called a path from  $v_I$  to  $v_T$ .

**Definition 3.4** A parallel control-flow net (PCFN) of a concurrent logic program is an 8-tuple  $(V, V_{and}, V_{or}, A_c, A_{and}, A_{or}, s, T)$ , where  $(V, A_c, A_{and}, A_{or})$  is a simple arc-classified digraph such that  $V = \{v \mid v \text{ represents a literal of the program}\}$ ,  $V_{and} \subset V$  is a set of vertices, called AND-parallel execution vertices,  $V_{or} \subset V (V_{or} \cap V_{and} = \emptyset)$  is a set of vertices, called OR-parallel execution vertices,  $A_c \subseteq V \times V, A_{and} \subseteq V_{and} \times V, A_{or} \subseteq V_{or} \times V$ ,  $s \in V$  is a unique vertex, called start vertex, such that  $in-degree(s) = 0$ ,  $T \subset V$  is a set of vertices, called termination vertices, such that for any  $t \in T$   $out-degree(t) = 0$  and  $t \neq s$ , and for any  $v \in V (v \neq s, v \in T)$ ,  $\exists t \in T$  such that there exists at least one path from  $s$  to  $v$  and at least one path from  $v$  to  $t$ . Any

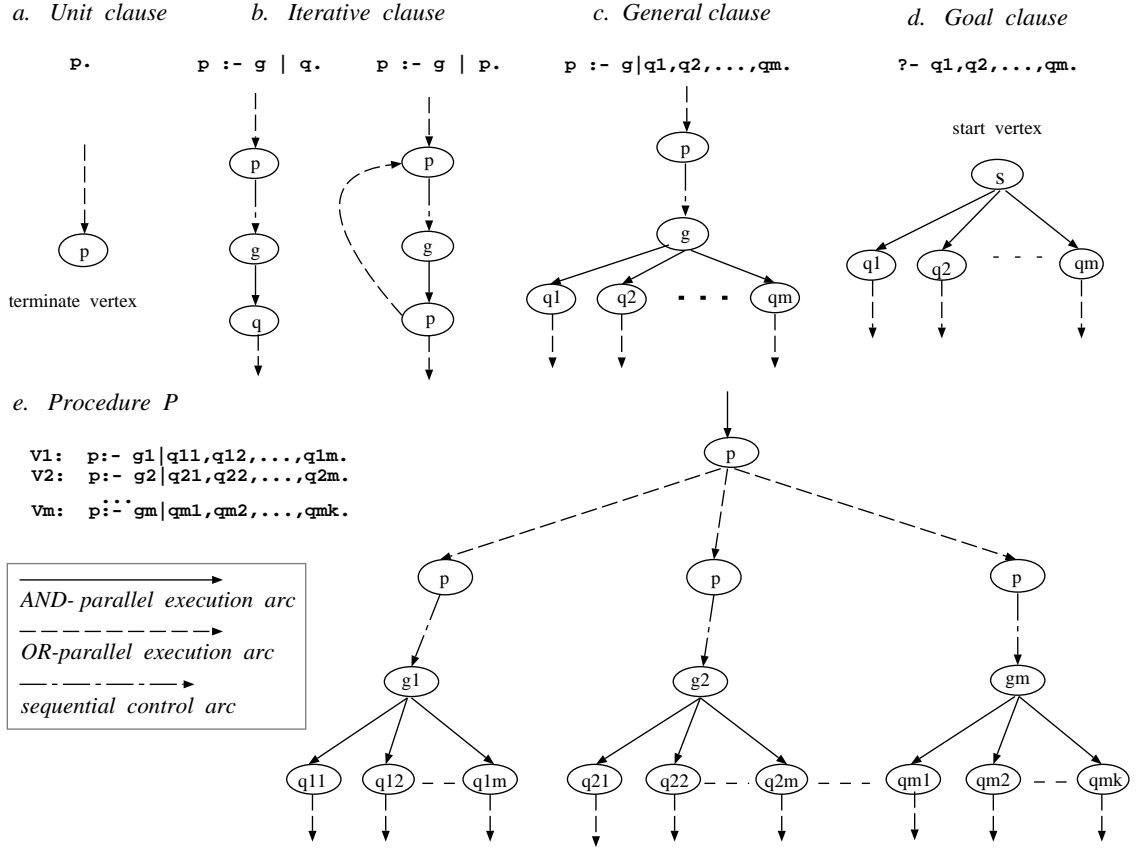


Figure 2: Transformation rules.

*arc*  $(v1, v2) \in A_c$  is called a sequential control arc, any *arc*  $(v1, v2) \in A_{and}$  is called an AND-parallel execution arc, and any *arc*  $(v1, v2) \in A_{or}$  is called an OR-parallel execution arc.

A PCFN can be used to represent multiple control flows in a concurrent logic program written in a concurrent logic programming language such as KL1, FCP, FGHC, and Flat Parlog.

In order to represent a concurrent logic program by a PCFN, we use vertices in the PCFN to represent the head literals, the body literals, and the guards where AND-parallel execution vertices represent the head literals (if the guards are empty) or guards of the clauses, and OR-parallel execution vertices represent the body literals, and use arcs to represent multiple control flows between literals.

Figure 2 shows some informal transformation rules for constructing the PCFN of a concurrent logic program. Having these transformation rules, we can transform a concurrent logic program into its PCFN as follows. First, we generate the sub-



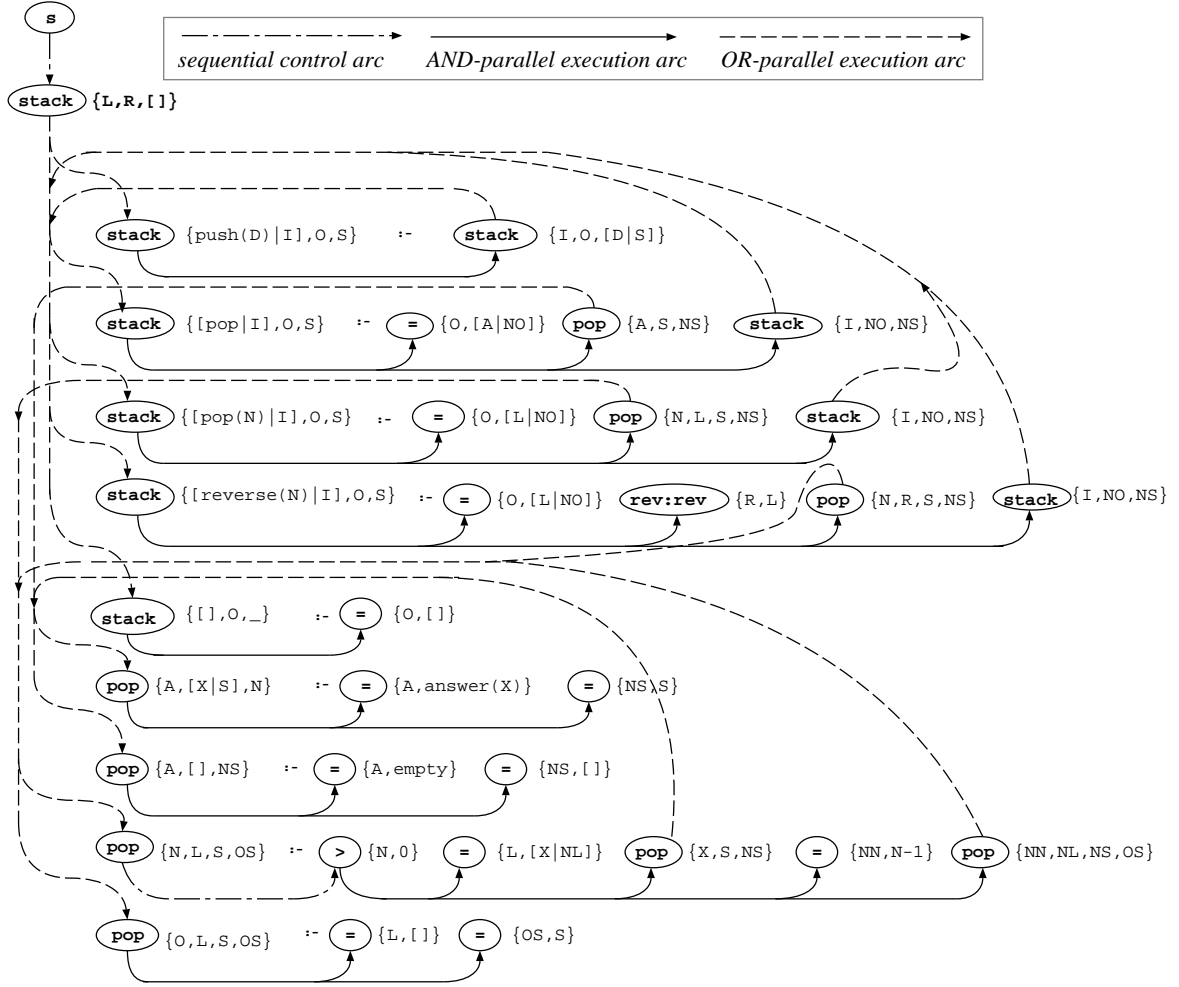


Figure 3: The APCFN of the program in Figure 1.

PCFNs for the goal and each clause in the program, then generate the sub-PCFNs for each procedure. Finally, we combine these subnets to form the complete PCFN by connecting all subnets of procedures using OR-parallel execution arcs.

In order to formally define program dependences between arguments in clauses of a concurrent logic program, we need an explicit representation of argument information of the program. We define the augmented parallel control-flow net to represent argument information.

**Definition 3.5** An augmented parallel control-flow net (APCFN) of a concurrent logic program is a 3-tuple  $(N_{PCFN}, \Sigma_A, \mathbf{A})$ , where  $N_{PCFN} = (V, V_{and}, V_{or}, A_c, A_{and}, A_{or}, s, T)$  is a PCFN of the program,  $\Sigma_A$  is a finite set of argument positions,  $\mathbf{A} : V \rightarrow P(\Sigma_A)$  is a partial function from  $V$  to the power set of  $\Sigma_A$  such that for any  $v \in V$ ,

$\mathbf{A}(v)$  is a set of all argument positions of  $v$ .

Note that an APCFN of a concurrent logic program can be regarded as a PCFN with argument information of all literals in the program. Therefore the APCFN can be used to represent not only control flows but also the argument information in the program. Figure 3 shows the APCFN of the program in Figure 1 which corresponds to a goal `stack(L,R,[])`.

#### 4. Primary Program Dependences and Argument Dependence Net

In this section we formally define three types of primary program dependences between arguments in concurrent logic programs based on the APCFN and also define a dependence-based representation for concurrent logic programs to explicitly represent the three types of primary program dependences in the programs.

##### 4.1. Mode Analysis of Concurrent Logic Programs

In order to explicitly represent data flows in a concurrent logic program, we should capture information concerning reading and writing of variables in the program. Such information are called *mode information*. Mode information of a literal in a concurrent logic program is an assertion about which of its arguments are input arguments, and which are output arguments, in any call to that literal arising from that program. Intuitively, an input or output argument mode indicates whether a data value is required as input or will be produced as output. The process to infer mode information of a program is called *mode analysis*. Mode information has many applications in the generation of efficient code for logic programs. For concurrent logic programs, mode information can be used to optimize code generated for argument matching to avoid suspensions, and also used to partition a concurrent logic program into threads of higher granularity for more efficient execution [11, 12]. In this paper, we use mode information to define primary program dependences between arguments in a concurrent logic program. Algorithms for analyzing mode information of sequential logic programs have been proposed in [3, 5]. For concurrent logic programs, Ueda and Morita [20] proposed a mode analysis algorithm based on the representation of procedure paths and their relationships as rooted graphs (“rational trees”). Their algorithm gives modes to any path of all predicates in a concurrent logic program. Krishna Rao *et al.* [13] proposed another mode analysis algorithm. Their algorithm gives modes to any argument positions of clauses in a concurrent logic program. Since in this paper the definition of program dependences between arguments in a concurrent logic program needs only mode information of argument positions of clauses in the program, we adopt the mode analysis algorithm proposed by Krishna Rao *et al.* [13] for our purpose. Due to the limitation of spaces we will not introduce the algorithm here. The details of the algorithm of how to analyze the mode information of argument

positions can be found in [13].

In the following we assume that the mode information of each argument position in a concurrent logic program has been inferred by the algorithm proposed by Krishna Rao *et al.*, and has the type, *in* or *out* [13]. We use  $\mathbf{M}(\alpha)$  to represent the mode of an argument  $\alpha$  and  $\Gamma(\alpha)$  to represent the set of all variables which appear in an argument  $\alpha$  of the program.

#### 4.2. Primary Program Dependences

In a concurrent logic program, data can be transferred in two ways: either within a clause between two arguments sharing a common variable, or from one clause to another via unification. If we know the mode information of arguments of the program, we can further determine the direction that data is transferred. To represent such information in a concurrent logic program, we introduce two types of primary program dependences named *sharing dependence* which reflects data flows in a single clause due to sharing variables, and *communication dependence* which reflects data flows in a single clause due to interprocess communications. Moreover, according to the direction which data is transferred and the mode information of the arguments in a clause, we can further divide the sharing dependences into two categories: *backward-sharing dependences* which reflect the data-flow in a single clause from an argument of a head literal to an argument of a guard test predicate, or a body literal, and *forward-sharing dependences* which reflect the data-flow in a single clause from an argument of a guard test predicate or a body literal to an argument of a head literal. We give the formal definitions of these primary program dependences based on the augmented parallel control-flow net and mode information of a concurrent logic program.

**Definition 4.1** Let  $N_{APCFN} = (N_{PCFN}, \Sigma_A, \mathbf{A})$  be the APCFN of a concurrent logic program  $P$ , where  $N_{PCFN} = (V, V_{and}, V_{or}, A_c, A_{and}, A_{or}, s, T)$  is the PCFN of  $P$ . Let  $u, v \in V$  be two vertices of  $N_{APCFN}$  such that  $u$  represents a body literal or a guard test predicate and  $v$  represents a head literal of a clause  $C$  of  $P$ , and  $\alpha \in \mathbf{A}(u)$  and  $\alpha' \in \mathbf{A}(v)$  be two argument positions such that  $\alpha = (C, j, u, k)$  and  $\alpha' = (C, 0, v, k')$ .  $\alpha$  is backward sharing-dependent on  $\alpha'$  iff all of the following conditions hold:

1.  $\Gamma(\alpha) \cap \Gamma(\alpha') \neq \emptyset$ , i.e., there is at least a variable shared by  $\alpha$  and  $\alpha'$ ,
2.  $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = \text{in}$ , and
3. there exists a path from  $v$  to  $u$  in  $N_{APCFN}$ .

For example, in clause  $C4$  of the program in Figure 1, the argument denoted by the input position  $(C4, 3, \text{pop}, 2)$  is backward sharing-dependent on the argument denoted by the input position  $(C4, 0, \text{stack}, 2)$  since there is a shared variable  $S$  between these two arguments and also exists a path from the head literal `stack(reverse(N)|I], 0, S)` to the fourth literal `pop(N, R, S, NS)` of clause  $C4$  in the

corresponding APCFN. Backward-sharing dependences are shown in Figure 4 with thin solid arcs.

**Definition 4.2** Let  $N_{APCFN} = (N_{PCFN}, \Sigma_A, \mathbf{A})$  be the APCFN of a concurrent logic program  $P$ , where  $N_{PCFN} = (V, V_{and}, V_{or}, A_c, A_{and}, A_{or}, s, T)$  is the PCFN of  $P$ . Let  $u, v \in V$  be two vertices of  $N_{APCFN}$  such that  $u$  represents a head literal and  $v$  represents a body literal or a guard test predicate, and  $\alpha \in \mathbf{A}(u)$  and  $\alpha' \in \mathbf{A}(v)$  be two argument positions such that  $\alpha = (C, 0, u, k)$  and  $\alpha' = (C, j, v, k')$ .  $\alpha$  is forward sharing-dependent on  $\alpha'$  iff all of the following conditions hold:

1.  $\Gamma(\alpha) \cap \Gamma(\alpha') \neq \phi$ , i.e., there is at least a variable shared by  $\alpha$  and  $\alpha'$ ,
2.  $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = \text{out}$ , and
3. there exists a path from  $v$  to  $u$  in  $N_{APCFN}$ .

For example, in clause  $C8$  of the program in Figure 1, the argument denoted by the output position  $(C8, 0, \text{pop}, 4)$  is forward sharing-dependent on the argument denoted by the output position  $(C8, 5, \text{pop}, 4)$  since there is a shared variable  $\text{OS}$  between these two arguments and also exists a path from the head literal  $\text{pop}(\text{N}, \text{L}, \text{S}, \text{OS})$  to the fifth literal  $\text{pop}(\text{NN}, \text{NL}, \text{NS}, \text{OS})$  of clause  $C8$  in the corresponding APCFN. Forward-sharing dependences are shown in Figure 4 with the same thin solid arcs as backward-sharing dependences.

Based on Definitions 4.1 and 4.2, a sharing dependence between two arguments in a clause can be defined as a backward-sharing dependence or a forward-sharing dependence.

Notice that our definition of a sharing dependence in a single clause requires that, in addition to a shared variable between two arguments in the clause, a path from one argument to another in the corresponding APCFN must exist also. As a result, our definition excludes the case that two arguments, which belong to two body literals, share a common variable. This case will be handled in the following, and referred to a new type of primary program dependences, the *communication dependences*. The communication dependences also reflect the data flow in a single clause, and more importantly, reflect the fact of interprocess communications between two body literals.

**Definition 4.3** Let  $N_{APCFN} = (N_{PCFN}, \Sigma_A, \mathbf{A})$  be the APCFN of a concurrent logic program  $P$ , where  $N_{PCFN} = (V, V_{and}, V_{or}, A_c, A_{and}, A_{or}, s, T)$  is the PCFN of  $P$ . Let  $u, v \in V$  be two vertices of  $N_{APCFN}$  that represent two body literals of a clause  $C$  of  $P$ , and  $\alpha \in \mathbf{A}(u)$  and  $\alpha' \in \mathbf{A}(v)$  be two argument positions such that  $\alpha = (C, j, u, k)$  and  $\alpha' = (C, j', v, k')$ .  $\alpha$  is communication-dependent on  $\alpha'$  iff all of the following conditions hold:

1.  $\Gamma(\alpha) \cap \Gamma(\alpha') \neq \phi$ , i.e., there is at least a variable shared by  $\alpha$  and  $\alpha'$ ,

2.  $\mathbf{M}(\alpha) = in$  and  $\mathbf{M}(\alpha') = out$ ,
3. there exists no path between  $u$  and  $v$  in  $N_{APCFN}$ , and
4. for any  $\alpha'' \in \mathbf{A}(w)$  where  $w \in V$ , there exists no sharing dependence between  $\alpha''$  and  $\alpha$ , and also between  $\alpha''$  and  $\alpha'$ .

Intuitively, a communication dependence exists between two arguments which belong to two body literals in a clause that share a common variable such that the variable does not appear in the head literal of the clause.

For example, in clause  $C8$  of the program in Figure 1, the argument denoted by the input position  $(C8, 3, pop, 2)$  is communication-dependent on the argument denoted by the output position  $(C8, 5, pop, 2)$  since there is a shared variable  $NS$  between these two arguments which belong to two body literals  $pop(X, S, NS)$  and  $pop(NN, NL, NS, OS)$  and  $NS$  does not appear in the head literal  $pop(N, L, S, OS)$  of  $C8$ . Communication dependences are shown in Figure 4 with thick dashed arcs.

Data as we motioned above, can be transferred not only in a single clause but also between different clauses via unifications in a concurrent logic programs. The third type of primary program dependences named *unification dependences* are therefore introduced to capture such information and reflect the data flow between arguments in different clauses in a concurrent logic program. Similar to sharing dependences, according to the direction which data is transferred and the mode information of the arguments between two clauses, we also divide the unification dependences into two categories: one which is called *backward-unification dependences* to reflect the data-flow from an input argument position of a head literal of a clause to an input argument position of a body literal of another clause, and the other which is called *forward-unification dependences* to reflect the data-flow from an output argument position of a body literal of a clause to an output argument position of a head literal of another clause.

**Definition 4.4** Let  $N_{APCFN} = (N_{PCFN}, \Sigma_A, \mathbf{A})$  be the APCFN of a concurrent logic program  $P$ , where  $N_{PCFN} = (V, V_{and}, V_{or}, A_c, A_{and}, A_{or}, s, T)$  is the PCFN of  $P$ . Let  $u, v \in V$  be two vertices of  $N_{APCFN}$  such that  $u$  represents a head literal of a clause  $C$  of  $P$  and  $v$  represents a body literal of a clause  $C'$  of  $P$ , and  $\alpha \in \mathbf{A}(u)$  and  $\alpha' \in \mathbf{A}(v)$  be two argument positions such that  $\alpha = (C, 0, p, k)$  and  $\alpha' = (C', j', p, k)$ .  $\alpha$  is backward unification-dependent on  $\alpha'$  iff all of the following conditions hold:

1.  $(u, v) \in A_{or}$  is an OR-parallel execution arc, and
2.  $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$ .

For example, in the program in Figure 1, the argument denoted by the input position  $(C8, 0, pop, 3)$  of clause  $C8$  is unification-dependent on the argument denoted by the input position  $(C4, 3, pop, 3)$  of clause  $C4$  since there exists a possible unification of the fourth literal  $pop(N, R, S, NS)$  of  $C4$  and the head literal  $pop(N, L, S, OS)$  of  $C8$ . backward-unification dependences are shown in Figure 4 with thin dashed arcs.

**Definition 4.5** Let  $N_{APCFN} = (N_{PCFN}, \Sigma_A, \mathbf{A})$  be the APCFN of a concurrent logic program  $P$ , where  $N_{PCFN} = (V, V_{and}, V_{or}, A_c, A_{and}, A_{or}, s, T)$  is the PCFN of  $P$ . Let  $u, v \in V$  be two vertices of  $N_{APCFN}$  such that  $u$  represents a body literal of a clause  $C$  of  $P$  and  $v$  represents a head literal of a clause  $C'$  of  $P$ , and  $\alpha \in \mathbf{A}(u)$  and  $\alpha' \in \mathbf{A}(v)$  be two argument positions such that  $\alpha = (C, j, p, k)$  and  $\alpha' = (C', 0, p, k)$ .  $\alpha$  is forward unification-dependent on  $\alpha'$  iff all of the following conditions hold:

1.  $(u, v) \in A_{or}$  is an OR-parallel execution arc, and
2.  $\mathbf{M}(\alpha') = \mathbf{M}(\alpha) = out$ .

For example, in the program in Figure 1, the argument denoted by the output position  $(C4, 3, pop, 2)$  of the clause  $C4$  is unification-dependent on the argument denoted by the output position  $(C8, 0, pop, 2)$  of clause  $C8$  since there exists a possible unification of the fourth literal  $pop(\mathbf{N}, \mathbf{R}, \mathbf{S}, \mathbf{NS})$  of  $C4$  and the head literal  $pop(\mathbf{N}, \mathbf{L}, \mathbf{S}, \mathbf{OS})$  of  $C8$ . Forward-unification dependences are shown in Figure 4 with the same thin dashed arcs as backward-unification dependences.

Based on Definitions 4.4 and 4.5, a unification dependence between two arguments in two different clauses can be defined as a backward-unification dependence or a forward-unification dependence.

#### 4.3. The Argument Dependence Net

As showed in [18], a dependence-based representation such as the *program dependence graph* (PDG) [6] for imperative programs, is well suited for computing slices of the programs since the representation provides a powerful framework for dependence analysis of the programs. In order to slice concurrent logic programs, we would like to use a similar representation to represent primary program dependences in a concurrent logic program explicitly. This motivates us to define the argument dependence net which is an arc-classified digraph to represent all primary program dependences in a concurrent logic program. Roughly speaking, the argument dependence net can be regarded as an extension of the PDG to the case of concurrent logic programs. The net has one vertex for each argument position in the program, and there is an arc in the argument dependence net for one of each type of primary program dependences between arguments.

**Definition 4.6** The Argument Dependence Net (ADN) of a concurrent logic program  $P$  is an arc-classified digraph  $(V_A, Sha, Com, Uni)$ , where:

- $V_A$  is the set of all arguments positions of  $P$ ;
- $Sha$  is the set of sharing dependence arcs such that any  $(\alpha, \alpha') \in Sha$  iff  $\alpha$  is backward or forward sharing-dependent on  $\alpha'$ ;
- $Com$  is the set of communication dependence arcs such that any  $(\alpha, \alpha') \in Com$  iff  $\alpha$  is communication-dependent on  $\alpha'$ ;

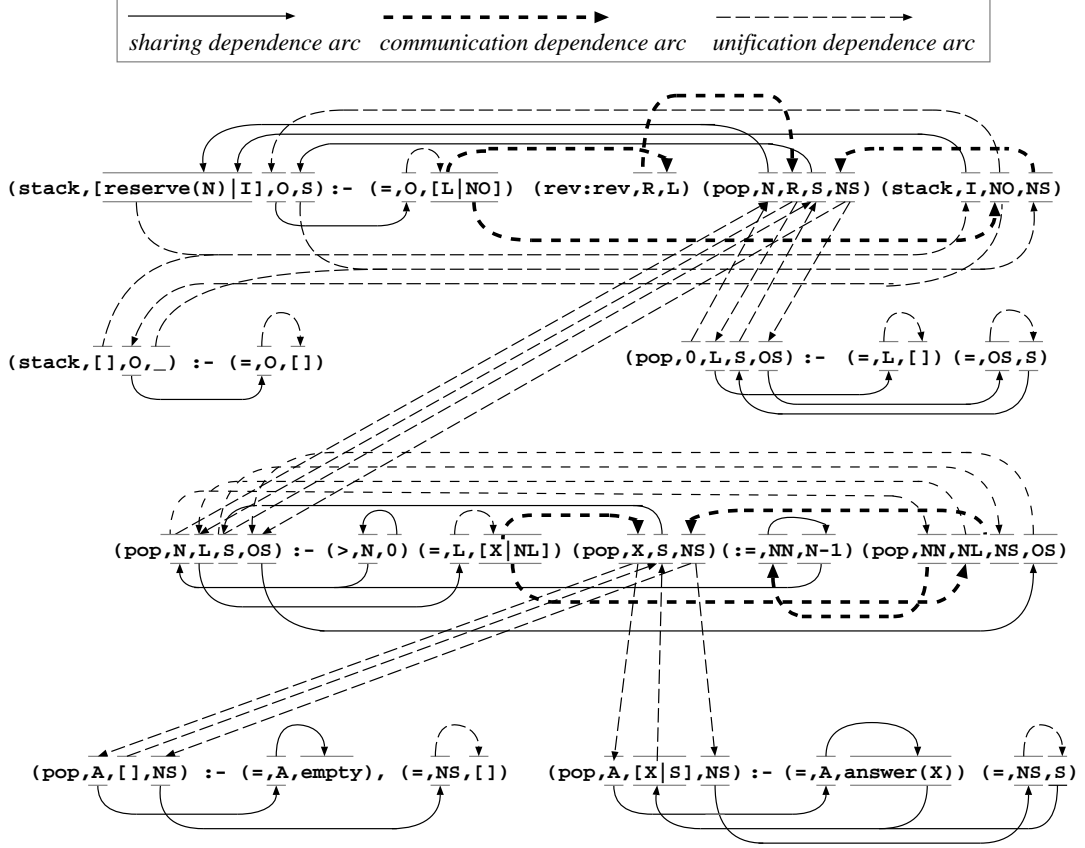


Figure 4: The partial ADN of the program in Figure 1.

- *Uni* is the set of unification dependence arcs such that any  $(\alpha, \alpha') \in Uni$  iff  $\alpha$  is backward or forward unification-dependent on  $\alpha'$ .

Figure 4 shows the ADN of the program in Figure 1. Because of the limitation of spaces, we only show a partial ADN of the program which is related to the fourth clause *C4* of the program. The other parts of the ADN of the program can be drawn easily and are omitted in the figure.

## 5. Computing Slices of Concurrent Logic Programs

The notions of static slices introduced in Section 2, give only some general views of slicing of concurrent logic programs, and do not show how to compute them. In this section, we refine those notions based on the ADN of concurrent logic programs.

**Definition 5.1** Let  $P$  be a concurrent logic program and  $N_{ADN} = (V_A, Sha, Com, Uni)$  be the ADN of  $P$ . A static slicing criterion for  $N_{ADN}$  is an argument position  $\alpha \in V_A$ .

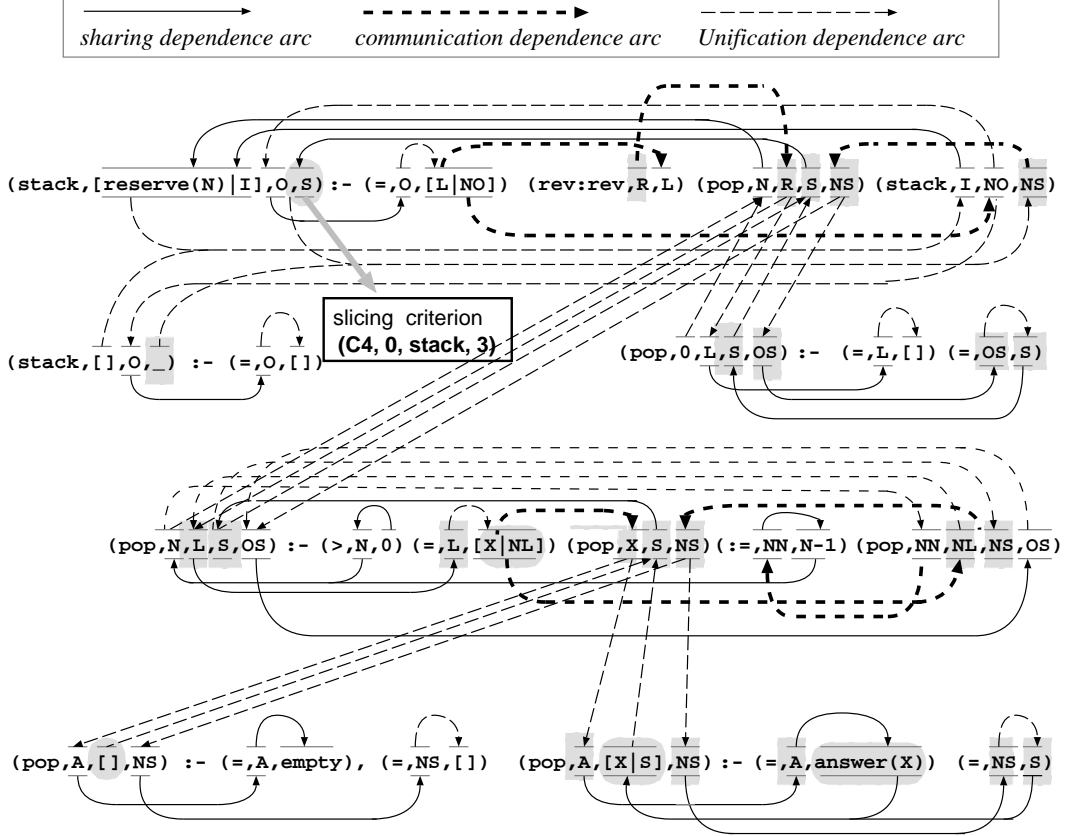


Figure 5: The partial ADN of the program in Figure 1 along with a slice over ADN.

**Definition 5.2** Let  $P$  be a concurrent logic program and  $N_{ADN} = (V_A, Sha, Com, Uni)$  be the ADN of  $P$ . The static backward slice  $SBS_N(\alpha)$  of  $N_{ADN}$  on a given static slicing criterion  $\alpha$  is a subset of vertices of  $V_A$ ,  $SBS_N(\alpha) \subseteq V_A$ , such that for any  $\alpha' \in V_A$ ,  $\alpha' \in SBS_N(\alpha)$  if and only if there exists a path from  $\alpha'$  to  $\alpha$  in the ADN.

**Definition 5.3** Let  $P$  be a concurrent logic program and  $N_{ADN} = (V_A, Sha, Com, Uni)$  be the ADN of  $P$ . The static forward-slice  $SFS_N(\alpha)$  of  $N_{ADN}$  on a given static forward-slicing criterion  $\alpha$  is a subset of vertices of  $V_A$ ,  $SFS_N(\alpha) \subseteq V_A$ , such that for any  $\alpha' \in V_A$ ,  $\alpha' \in SFS_N(\alpha)$  if and only if there exists a path from  $\alpha$  to  $\alpha'$  in the ADN.

Notice that in term of Definitions 5.2 and 5.3, a static backward or forward slice over the ADN of a concurrent logic program, is a set of vertices of the ADN. However, since our aim is to obtain a slice of a program, we should map a vertex in the ADN to a reduced literal of the program. Since a vertex in the ADN which represents an argument position of the program has already contained the predicate name of the literal, we can get such mapping quite easily.



```

C1: stack([push(D)|I],O,S):-
    stack(I,O,[D|S]).
C2: stack([pop|I],O,S):-
    O=[A|NO], pop(A,S,NS), stack(I,NO,NS).
C3: stack([pop(N)|I],O,S):-
    O=[L|NO], pop(N,L,S,NS), stack(I,NO,NS).
C4: stack(_ , _ , S) :-
    O=[L|NO], rev:rev(R,_), pop(_ ,R,S,NS), stack(_ , _ , NS)
C5: stack(_ , _ , _):-
    O=[ ].
C6: pop(_ , [X|S] , _):-
    A=answer(X), NS=S.
C7: pop(_ , [ ] , _):-
    A=empty, NS=[ ].
C8: pop(_ , L , S , _):-
    N>0 | L=[X|NL], pop(X,S,NS), NN=N-1, pop(_ , NL,NS, _).
C9: pop(_ , _ , S , OS):-
    L=[ ], OS=S.

```

slicing criterion  
 (stack([reverse(N)|I], O, S), S)

Figure 6: A static backward slice of the program in Figure 1.

**Definition 5.4** Let  $P$  be a concurrent logic program and  $N_{ADN} = (V_A, Sha, Com, Uni)$  be the ADN of  $P$ . Let  $S_N$  be a static backward or forward slice over the  $N_{ADN}$ , and  $\Sigma_L$  be a set of reduced literals of  $P$ ,  $\beta : S_N \rightarrow \Sigma_L$  is a function from  $S_N$  to  $\Sigma_L$  such that for any argument position  $\alpha \in S_N$ ,  $\beta(\alpha) \in \Sigma_L$  is a reduced literal of  $P$  corresponding to  $\alpha$  iff  $\beta(\alpha)$  contains  $\alpha$ .

Having the mapping function  $\beta$  defined above, a static backward or forward slice of a concurrent logic program, which is a set of reduced literals, can be obtained by the mapping function.

Figure 5 shows the partial ADN of the program **STACK** in Figure 1 and a static backward slice over the ADN. The slice is represented in shaded vertices in the net and is computed with respect to the slicing criterion  $\alpha = (C4, 0, stack, 3)$ . Figure 6 shows the program **STACK** in Figure 1 and a static backward slice of the program with the slicing criterion  $(stack([reverse(N)|I], O, S), S)$  corresponding to the head literal **stack([reverse(N)|I], O, S)** of clause  $C4$  of the program in Figure 1. The slice is represented in shaded reduced literals and obtained from the corresponding slice over the ADN in Figure 5 according to the mapping function defined in Definition 5.4.

## 6. Conclusions

We have discussed the problem of slicing concurrent logic programs. To solve this problem, we proposed three types of primary program dependences between arguments in concurrent logic programs and also presented the *argument dependence net*(ADN) for concurrent logic programs to explicitly represent these primary program dependences in the programs. Based on the ADN, our slicing algorithm can produce static slices for concurrent logic programs at argument level. Although here we presented the approach in term of KL1, a simple concurrent logic language, other versions for this approach for other concurrent logic programming languages are easily adaptable because they share their basic execution mechanisms with KL1.

Program slicing has been widely studied for imperative programs during the last decade, however, for logic programs it is just starting. There are still many problems that should be solved in the area of slicing logic programs. First, since all existing definitions for slices of logic programs can be regarded as some extensions of imperative programs, the problems arose are: "What really is a slice of logic programs?" and "Are there some intrinsic differences between a slice of imperative programs and logic programs?" Second, since dynamic slicing usually produces more precise slices than static slicing because it considers only a particular execution of a program, "how to slice logic programs dynamically?" should be investigated. Third, current application areas for slicing of logic programs are completely the same as those for imperative programs, but "could we exploit some novel application areas of slicing that are specially useful in development of logic programs?" Finally, the static slices defined in this paper are not necessary executable, so "how to compute an executable slice for concurrent logic programs?" Moreover, to build a theoretical basis for dependence analysis of concurrent logic programs, it is necessary to give a formal semantics of sharing, communication and unification dependences as well as the argument dependence net for concurrent logic programs.

To demonstrate the usefulness of our slicing approach, we have developed a slicer for KL1 programs and embedded it into a support environment for program analysis of concurrent logic programs that is being developed [23] to aid bug location, code understanding, maintenance and complexity measurement. The next step for us is to perform some experiments to evaluate the usefulness of slicing in practical development of concurrent logic programs.

## Acknowledgements

We would like to thank Stéphane Schoenig and the anonymous referees for their valuable suggestions and comments on eariler drafts of the paper.

## References

- [1] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.
- [2] J. M. Bieman, L. M. Ott, "Measuring Functional Cohesion," *IEEE Transaction on Software Engineering*, Vol.20, No.8, pp.644-657, 1994.
- [3] J. Boye, J. Paakki, and J. Maluszyński, "Synthesis of Directionality Information for Functional Logic Programs," *Proc. Third International Workshop on Static Analysis*, LNCS 72 4, Springer-Verlag, pp.165-177, 1993.
- [4] J. Cheng, "Slicing Concurrent Programs – A Graph-Theoretical Approach," *Proc. First International Workshop on Automated Algorithmic Debugging, Lecture Notes in Computer Science*, Vol.749, pp.223-240, Springer-Verlag, May, 1993.
- [5] S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs," *ACM Transaction on Programming Language and System*, Vol.11, No.3, pp.418-450, 1987.
- [6] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.
- [7] K. B. Gallagher, J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.
- [8] T. Gyimóthy, J. Paakki, "Static Slicing of Logic Programs," *Local proc. 2nd International Workshop on Automated and Algorithmic Debugging*, France, May, 1995.
- [9] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.
- [10] M. Kamkar, N. Shahmehri, P. Fritzson, "Bug Localization by Algorithmic Debugging and Program Slicing," *Proc. International Workshop on Programming Language Implementation and Logic Programming, Lecture Notes in Computer Science*, Vol.456, pp.60-74, Springer-Verlag, 1990.
- [11] A. King, P. Soper, "Schedule Analysis of Concurrent Logic Programs," *Proc. International Joint Conference and Symposium on Logic Programming*, pp.478-492, MIT Press, 1992.

- [12] M. Korsloot, E. Tick, "Sequentializing Parallel Programs," *Proc. Phoenix Seminar and Workshop on Declarative Programming*, Hohrirt, Sasbachwalden, Germany. Springer-Verlag, 1991.
- [13] M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar, "Proving Termination of GHC Programs," *Proc. Tenth International Conference on Logic Programming*, pp.720-736, MIT Press, 1993.
- [14] K. J. Ottenstein, L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.
- [15] S. Schoenig, M. Ducassé, "A Hybrid Backward Slicing Algorithm Producing Executable Slices for Prolog," *Proc. ILPS'95 Workshop on Logic Programming Environments*, Portland, Oregon, USA, December 8, 1995.
- [16] E. Shapiro, "The Family of Concurrent logic Programming Languages," *ACM Computing Surveys*, Vol. 21, No. 3, pp.412-510, September, 1989.
- [17] T. Takeuchi, K. Furukawa, "Parallel Logic Programming Languages," in E. Shapiro (ed.) *Concurrent Prolog: Collected Papers*, pp.188-201, Vol. 1, MIT press, 1987.
- [18] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, Vol.3, No.3, pp.121-189, September, 1995.
- [19] K. Ueda, T. Chikayama, "Design of the Kernel Language for the Parallel Inference Machine," *The Computer Journal*, Vol.33, No.6, pp.494-500, 1990.
- [20] K. Ueda, M. Morita, "Moded Flat GHC and Its Message-Oriented Parallel Implementation Technique," *New Generation Computing*, Vol.13, No.1, pp.3-43, 1994.
- [21] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.
- [22] J. Zhao, J. Cheng, and K. Ushijima, "Literal Dependence Net and Its Use in Concurrent Logic Programming Environment", *Proc. Workshop on Parallel Logic Programming attached to FGCS'94*, pp.127-141, Tokyo, Japan, December, 1994.
- [23] J. Zhao, J. Cheng, and K. Ushijima, "Program Dependence Analysis of Concurrent Logic Programs and Its Applications", in *Proc. International Conference on Parallel and Distributed Systems (ICPADS'96)*, pp.282-291, Tokyo, Japan, June 3-6, 1996, IEEE Computer Society Press.